SV Randomization & Functional Coverage Assignment

Abdelrahman Mohamed Ali

Gmail	abdelrahmansalby23@gmail.com
LinkedIn	Abdelrahman Mohamed
GitHub	Abdelrahman1810

verification Plane flow

Label	Description	Stimulus Generation	Functional Coverage	Functionality Check
ALSU_1	Incase of invalid cases do not occur, when opcode is add, then out should perform the addition on ports A and B taking cin if parameter FULL_ADDER is high	Randomization under constraints on the A and B to have the maximum, minimum and zero values most of the time	Included as coverpoint for A and B. Included with cross coverage when ALU opcode is addition or multiplication	Output Checked against golden model
ALSU_2	Incase of invalid cases do not occur, when opcode is mult, then out should perform the multiplication on ports A and B	Randomization under constraints on the A and B to have the maximum, minimum and zero values most of the time	Included in coverpoint2 for A and B. Included with cross coverage when ALU opcode is addition or multiplication	Output Checked against golden model
ALSU_3	When invalid cases exist, out output should be low and leds should blink	Randomization under constraints where invalid cases do not occur as frequent as valid cases	Included in a coverpoint for opcode. Included with cross coverage to make sure invalid cases occur	Output Checked against golden model
ALSU_4	If invalid cases do not occur and the bypass inputs are high, then the output out should by bypass port A or B based on the prioirty if the both bypass ports are high	Randomization for bypass	Included in a coverpoint for bypass	Output Checked against golden model
ALSU_5	If invalid cases do not occur and the reduction operation are high and opcode nither OR nor XOR, then the output out should by zero and leds should blink as an INVALED case	Randomization for reduction operation	Included in cover group a cross covarage for reduction Invalid (reduction_invalid) Included with cross coverage when ALU opcode is addition or multiplication	Output Checked against golden model and from invvalid signal
ALSU_6	Incase of invalid cases do not occur, when opcode is OR and reduction operation is low, then out should perform make A OR B	Randomization under constraints on the A to be invert B (A=~B)	Included in coverpoint for A and B. Included with cross coverage when ALU opcode is arithmatic	Output Checked against golden model
ALSU_7	Incase of invalid cases do not occur, when opcode is XOR and reduction operation is low, then out should perform A XOR B	Randomization under constraints on the A to be invert B (A=~B)	Included in coverpoint for A and B. Included with cross coverage when ALU opcode is arithmatic	Output Checked against golden model
ALSU_8	Incase of invalid cases do not occur, when opcode either OR or XOR and reduction operation is high for port A, then out should perform reduction OR or XOR on ports A onley	Randomization under constraint the input B most of the time to have one bit high in its 3 bits while constraining the A to be low	Included in coverpoint for A. when A is {001, 010, 100} and only the red_op_A is high	Output Checked against golden model
ALSU_9	Incase of invalid cases do not occur, when opcode either OR or XOR and reduction operation is high for port B, then out should perform reduction OR or XOR on ports B onley	Randomization under constraint the input A most of the time to have one bit high in its 3 bits while constraining the B to be low	Included in coverpoint for A. when B is {001, 010, 100} and onley red_op_A is low and red_op_B is high	Output Checked against golden model
ALSU_10	Incase of invalid cases do not occur, when opcode is SHIFT and direction is low, then out should perform shift left logical for last out with serial in	Randomization for A and B with no constraint	Included in coverpoint2 for A and B. Included with cross coverage when ALU opcode is SHIFT or ROTATE	Output Checked against golden model
ALSU_11	Incase of invalid cases do not occur, when opcode is SHIFT and direction is high, then out should perform shift right logical for last out with serial in	Randomization for A and B with no constraint	Included in coverpoint2 for A and B. Included with cross coverage when ALU opcode is SHIFT or ROTATE	Output Checked against golden model
ALSU_12	Incase of invalid cases do not occur, when opcode is ROTATE and direction is high, then out should perform the MSB to be LSB	Randomization for A and B with no constraint	Included in coverpoint2 for A and B. Included with cross coverage when ALU opcode is SHIFT or ROTATE	Output Checked against golden model
ALSU_13	Incase of invalid cases do not occur, when opcode is ROTATE and direction is low, then out should perform the LSB to be MSB	Randomization for A and B with no constraint	Included in coverpoint2 for A and B. Included with cross coverage when ALU opcode is SHIFT or ROTATE	Output Checked against golden model
ALSU_14	rst is high	Randomization under constraints on rst to be low most of time and make the first randomized value is high in post_randomize	include in coverpoint with weight zero	Output Checked from wave
ALSU_15	making opcode always valid	randomize an array and make it unique with no INVALID value	included in coverpoint ALU_cp	Output Checked against golden model

Do file

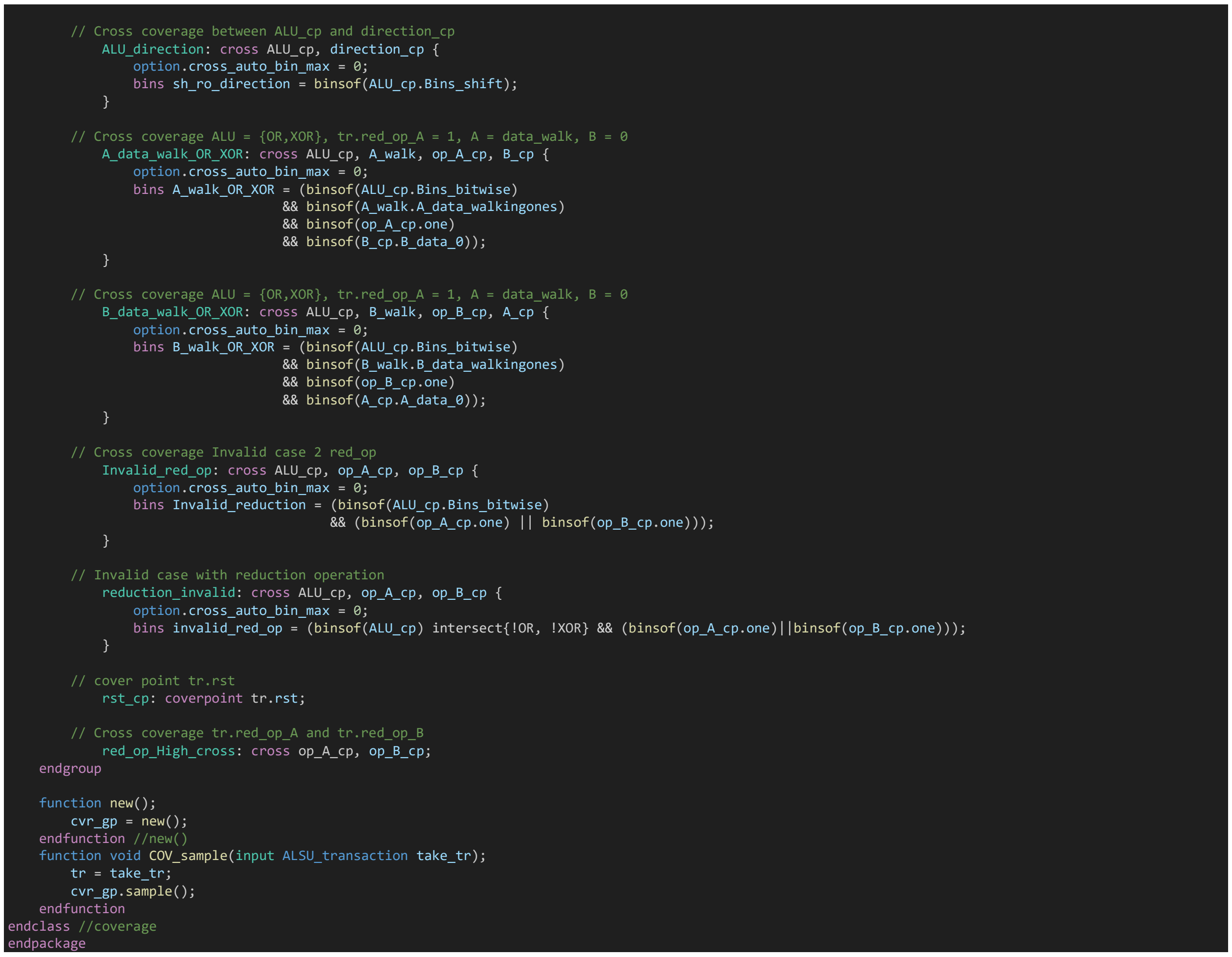
```
vlib work

vlog -coveropt 3 +cover +acc {Codes/pkgs/shared_pkg.sv}
vlog -coveropt 3 +cover +acc {Codes/pkgs/transaction_pkg.sv}
vlog -coveropt 3 +cover +acc {Codes/pkgs/coverage_pkg.sv}
##
vlog -coveropt 3 +cover +acc {Codes/Design/ALSU.v}
vlog -coveropt 3 +cover +acc {Codes/Design/ALSU_sva.sv}
##
vlog -coveropt 3 +cover +acc {Codes/golden_model/ALUS_ref.sv}
##
vlog -coveropt 3 +cover +acc {Codes/testbench/testbench.sv}

vsim -voptargs=+acc work.testbench
add wave *

add wave /testbench/assert_invaled
add wave /testbench/assert_invaled_opcode
add wave /testbench/assert_invaled_ref_op
add wave /testbench/assert_out
add wave /testbench/sva/redA_OR_assert
add wave /testbench/sva/redB_OR_assert
add wave /testbench/sva/redA_XR_assert
add wave /testbench/sva/redB_XR_assert
add wave /testbench/sva/Invalid_assert
add wave /testbench/sva/OR_assert
add wave /testbench/sva/XOR_assert
add wave /testbench/sva/add_assert
add wave /testbench/sva/mult_assert
add wave /testbench/sva/shiftL_assert
add wave /testbench/sva/shiftR_assert
add wave /testbench/sva/rotateL_assert
add wave /testbench/sva/rotateR_assert

add wave /testbench/sva/reset_assertion/rst_out_assert
add wave /testbench/sva/reset_assertion/rst_leds_assert

# free thw next command and comment the quit -sim to see the wave form
#run -all

vsim -coverage -vopt work.testbench -c -do "coverage save -onexit -du ALSU -directive -codeAll cover.ucdb; run -all"
#
coverage report -detail -cvg -directive -comments -output {Reports\FUNCTION_COVER_ALSU.txt} {}
#
quit -sim
#
vcover report cover.ucdb -details -all -annotate -output  {Reports\cover_alsu.txt}
#
vcover report -html cover.ucdb -output  {Reports\html_report\.}
```

Shared_pkg:

```
package shared_pkg;
    parameter INPUT_PRIORITY = "A";
    parameter FULL_ADDER = "ON";

    typedef enum reg [2:0] {OR, XOR, ADD, MULT, SHIFT, ROTATE, INVALID_6, INVALID_7} opcode_e;
    parameter MAXPOS = 3;
    parameter ZERO = 0;
    parameter MAXNEG = -4;
    parameter RST_ACTIVATE = 5;
    parameter INVALID_ACTIVATE = 5;
    parameter BYPASS_ACTIVATE = 10;
    parameter REDACTION_ACTIVATE = 20;
    bit first_rst = 0;
    opcode_e valid_arr[] = {OR, XOR, ADD, MULT, SHIFT, ROTATE};
endpackage
```

coverage_pkg:

```
package coverage_pkg;
import shared_pkg::*;
import transaction_pkg::*;

class ALSU_coverage;
    ALSU_transaction tr = new();

    covergroup cvr_gp;
        // input A bins
            A_cp: coverpoint tr.A {
                bins A_data_0 = {ZERO};
                bins A_data_max = {MAXPOS};
                bins A_data_min = {MAXNEG};
                bins A_data_default = default;
                bins A_data_[] = {001, 010, 100};
            }

        // input B bins
            B_cp: coverpoint tr.B {
                bins B_data_0 = {ZERO};
                bins B_data_max = {MAXPOS};
                bins B_data_min = {MAXNEG};
                bins B_data_default = default;
                bins B_data_[] = {001, 010, 100};
            }

        // cover point for reduction operation red_op
            op_A_cp: coverpoint tr.red_op_A {
                bins one = {1};
                bins zero = {0};
            }
            op_B_cp: coverpoint tr.red_op_B {
                bins one = {1};
                bins zero = {0};
            }

        // Crossing to satsfied the data_walkingones of A and B
            A_walk: cross A_cp, op_A_cp {
                option.cross_auto_bin_max = 0;
                bins A_data_walkingones = binsof(A_cp.A_data_) && binsof(op_A_cp.one);
            }
            B_walk: cross B_cp, op_A_cp, op_B_cp {
                option.cross_auto_bin_max = 0;
                bins B_data_walkingones = (binsof(B_cp.B_data_)
                                        && binsof(op_A_cp.zero)
                                        && binsof(op_B_cp.one));
            }

        // cover point for tr.opcode (ALU)
            ALU_cp: coverpoint tr.opcode {
                bins Bins_shift[] = {SHIFT, ROTATE};
                bins Bins_arith[] = {ADD, MULT};
                bins Bins_bitwise[] = {OR, XOR};
                bins Bins_invalid = {INVALID_6, INVALID_7};
                bins Bins_trans = (OR => XOR => ADD => MULT => SHIFT => ROTATE);
            }

        // cover point for c_in
            cin_cp: coverpoint tr.cin;

        // cover point for tr.serial_in
            serial_cp: coverpoint tr.serial_in;

        // cover point for tr.direction
            direction_cp: coverpoint tr.direction;

        // Cross coverage between ALU_cp and A and B
            ALU_A: cross ALU_cp, A_cp {
                option.cross_auto_bin_max = 0;
                bins arith_permutations = binsof(ALU_cp.Bins_arith) && binsof(A_cp) intersect{ZERO, MAXPOS, MAXNEG};
            }
            ALU_B: cross ALU_cp, B_cp {
                option.cross_auto_bin_max = 0;
                bins arith_permutations = binsof(ALU_cp.Bins_arith) && binsof(B_cp) intersect{ZERO, MAXPOS, MAXNEG};
            }

        // Cross coverage between ALU_cp and cin_cp
            ALU_cin: cross ALU_cp, cin_cp {
                option.cross_auto_bin_max = 0;
                bins add_cin = binsof(ALU_cp) intersect{ADD};
            }

        // Cross coverage between ALU_cp and serial_cp
            ALU_serial: cross ALU_cp, serial_cp {
                option.cross_auto_bin_max = 0;
                bins shift_serial = binsof(ALU_cp) intersect{SHIFT};
            }
```



```
        // Cross coverage between ALU_cp and direction_cp
            ALU_direction: cross ALU_cp, direction_cp {
                option.cross_auto_bin_max = 0;
                bins sh_ro_direction = binsof(ALU_cp.Bins_shift);
            }

        // Cross coverage ALU = {OR,XOR}, tr.red_op_A = 1, A = data_walk, B = 0
            A_data_walk_OR_XOR: cross ALU_cp, A_walk, op_A_cp, B_cp {
                option.cross_auto_bin_max = 0;
                bins A_walk_OR_XOR = (binsof(ALU_cp.Bins_bitwise)
                                   && binsof(A_walk.A_data_walkingones)
                                   && binsof(op_A_cp.one)
                                   && binsof(B_cp.B_data_0));
            }

        // Cross coverage ALU = {OR,XOR}, tr.red_op_A = 1, A = data_walk, B = 0
            B_data_walk_OR_XOR: cross ALU_cp, B_walk, op_B_cp, A_cp {
                option.cross_auto_bin_max = 0;
                bins B_walk_OR_XOR = (binsof(ALU_cp.Bins_bitwise)
                                   && binsof(B_walk.B_data_walkingones)
                                   && binsof(op_B_cp.one)
                                   && binsof(A_cp.A_data_0));
            }

        // Cross coverage Invalid case 2 red_op
            Invalid_red_op: cross ALU_cp, op_A_cp, op_B_cp {
                option.cross_auto_bin_max = 0;
                bins Invalid_reduction = (binsof(ALU_cp.Bins_bitwise)
                                         && (binsof(op_A_cp.one) || binsof(op_B_cp.one)));
            }

        // Invalid case with reduction operation
            reduction_invalid: cross ALU_cp, op_A_cp, op_B_cp {
                option.cross_auto_bin_max = 0;
                bins invalid_red_op = (binsof(ALU_cp) intersect{!OR, !XOR} && (binsof(op_A_cp.one)||binsof(op_B_cp.one)));
            }

        // cover point tr.rst
            rst_cp: coverpoint tr.rst;

        // Cross coverage tr.red_op_A and tr.red_op_B
            red_op_High_cross: cross op_A_cp, op_B_cp;
    endgroup

    function new();
        cvr_gp = new();
    endfunction //new()
    function void COV_sample(input ALSU_transaction take_tr);
        tr = take_tr;
        cvr_gp.sample();
    endfunction
endclass //coverage
endpackage
```

transaction_pkg:

```
package transaction_pkg;
import shared_pkg::*;

class ALSU_transaction;
    rand opcode_e opcode;
    rand bit rst, cin, red_op_A, red_op_B, bypass_A, bypass_B, direction, serial_in;
    rand bit signed [2:0] A, B;
    bit [1:0]red_op_parameterTest;
    constraint rules1_7 {
        //rst constraint
            rst dist {0:=100-RST_ACTIVATE, 1:=RST_ACTIVATE};

        // Invalid cases constraint
            opcode dist {INVALID_6:=INVALID_ACTIVATE, INVALID_7:=INVALID_ACTIVATE, [0:5]:/100-2*INVALID_ACTIVATE};

        // A & B constraint when opcode is ADD or MULT
            (opcode == MULT || opcode == ADD) -> A dist {MAXPOS:=20, ZERO:=10, MAXNEG:=20, [MAXNEG+1:MAXPOS-1]:/50};
            (opcode == MULT || opcode == ADD) -> B dist {MAXPOS:=20, ZERO:=10, MAXNEG:=20, [MAXNEG+1:MAXPOS-1]:/50};

        // A & B constraint when opcode is OR or XOR and red_op_A is high
            ((opcode==XOR || opcode==OR) && red_op_A) -> A dist {3'b001:=30, 3'b010:=30, 3'b100:=30, [MAXNEG+1:MAXPOS-1]:/10};
            ((opcode==XOR || opcode==OR) && red_op_A) -> B == 0;

        // A & B constraint when opcode is OR or XOR and red_op_B is high
            ((opcode==XOR || opcode==OR) && red_op_B) -> A == 0;
            ((opcode==XOR || opcode==OR) && red_op_B) -> B dist {3'b001:=30, 3'b010:=30, 3'b100:=30, [MAXNEG+1:MAXPOS-1]:/10};

        //  Do not constraint the inputs A or B when the operation is shift or rotate
        // it's achieved by default after the 2,3 and 4 Constraint achieved

        // bypass constraint
            bypass_A dist {0:=100-BYPASS_ACTIVATE, 1:=BYPASS_ACTIVATE};
            bypass_B dist {0:=100-BYPASS_ACTIVATE, 1:=BYPASS_ACTIVATE};

        // red_op constraint
            red_op_A dist {0:=100-REDACTION_ACTIVATE, 1:=REDACTION_ACTIVATE};
            red_op_B dist {0:=100-REDACTION_ACTIVATE, 1:=REDACTION_ACTIVATE};

        // A & B constraint when opcode is OR or XOR and red_op is low
            ((opcode==XOR || opcode==OR) && ~red_op_A && ~red_op_B) -> A == ~B;
    }

    rand opcode_e arr [6];
    constraint rules_8 {
        foreach(arr[i])
            arr[i] inside {valid_arr};
        unique {arr};
    }

    function new();
    endfunction //new()
    function void post_randomize();
        if(!first_rst) rst = 1;
        first_rst = 1;
        if (red_op_parameterTest!=2'b11 && !rst && !bypass_A && !bypass_B) begin
            if (opcode==OR) begin
                red_op_A = 1;
                red_op_B = 1;
                red_op_parameterTest[0] = 1;
            end
            if (opcode==XOR) begin
                red_op_A = 1;
                red_op_B = 1;
                red_op_parameterTest[1] = 1;
            end
        end
    endfunction
endclass //ALSUtransaction
endpackage
```

Testbench code:

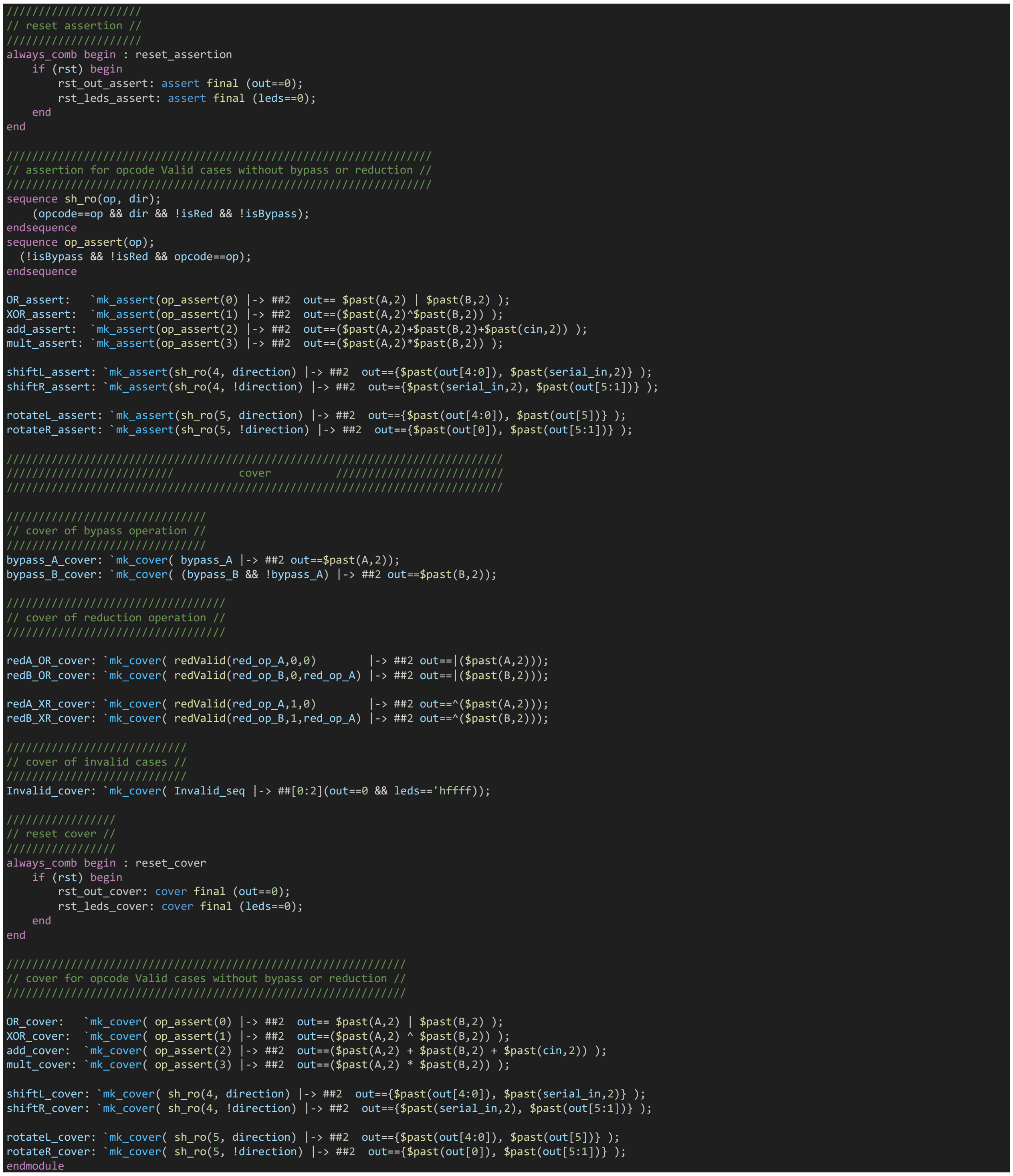
```
import shared_pkg::*;
import coverage_pkg::*;
import transaction_pkg::*;
`define reset disable iff(rst)
`define mk_assertion(sva_assert) assert property (@(posedge clk) disable iff(rst) (sva_assert))

module testbench ();
logic clk, rst, cin, red_op_A, red_op_B, bypass_A, bypass_B, direction, serial_in;
opcode_e opcode;
logic signed [2:0] A, B;
logic [15:0] leds, leds_ref;
logic [5:0] out, out_ref;
ALSU_transaction tr = new();
ALSU_coverage cov = new();

ALSU DUT(.*);
ALSU_sva sva(.*);
ALUS_ref REF(
    A, B, cin, serial_in, red_op_A, red_op_B,
    opcode, bypass_A, bypass_B, clk, rst, direction, leds_ref, out_ref
);

initial begin
    clk = 0;
    forever
        #1 clk = ~clk;
end

initial begin
    // First loop turn off constraint 8
    tr.rules_8.constraint_mode(0);
    repeat(500_000) begin
        randomization;
        @(negedge clk);
        sample;
    end

    // Forced rst
    bypass_A = 0; bypass_B = 0;
    red_op_A = 0; red_op_B = 0;
    rst = 1; @(negedge clk); rst = 0;

    // Seconed loop turn ON constraint 8 and turn OFF other constraints
    tr.rules1_7.constraint_mode(0);
    tr.rules_8.constraint_mode(1);
    repeat(10_000) begin
        randomization(1);
        foreach(tr.arr[i]) begin
            opcode = tr.arr[i];
            tr.opcode = tr.arr[i];
            @(negedge clk);
            sample;
        end
        $display("arr = %p",tr.arr);
    end
    $stop;
end

task randomization(bit con_rule8 = 0);
    assert(tr.randomize());
    if (!con_rule8) begin
        rst = tr.rst;
        bypass_A = tr.bypass_A;
        bypass_B = tr.bypass_B;
        red_op_A = tr.red_op_A;
        red_op_B = tr.red_op_B;
    end
    cin = tr.cin;
    direction = tr.direction;
    serial_in = tr.serial_in;
    opcode = tr.opcode;
    A = tr.A;
    B = tr.B;
endtask

task sample;
    if (~tr.rst||~bypass_A||~bypass_B) cov.COV_sample(tr);
endtask

assert_invaled:         assert property (@(posedge clk) disable iff(rst) REF.invalid |-> DUT.invalid);
assert_invaled_opcode:  assert property (@(posedge clk) disable iff(rst) REF.invalid_opcode |-> DUT.invalid_opcode);
assert_invaled_ref_op:  assert property (@(posedge clk) disable iff(rst) REF.invalid_red |-> DUT.invalid_red_op);
assert_out:             assert property (@(posedge clk) (out == out_ref));
endmodule
```

Design code:

```
////////////////////////////////////////////////////////////////////////////////
// Author: Kareem Waseem
// Course: Digital Verification using SV & UVM
//
// Description: ALSU Design
//
////////////////////////////////////////////////////////////////////////////////
module ALSU(A, B, cin, serial_in, red_op_A, red_op_B, opcode, bypass_A, bypass_B, clk, rst, direction, leds, out);
parameter INPUT_PRIORITY = "A";
parameter FULL_ADDER = "ON";
input clk, rst, cin, red_op_A, red_op_B, bypass_A, bypass_B, direction, serial_in;
input [2:0] opcode;
input signed [2:0] A, B;
output reg [15:0] leds;
output reg [5:0] out;

reg cin_reg, red_op_A_reg, red_op_B_reg, bypass_A_reg, bypass_B_reg, direction_reg, serial_in_reg;
reg [2:0] opcode_reg, A_reg, B_reg;

wire invalid_red_op, invalid_opcode, invalid;
assign invalid_red_op = (red_op_A_reg | red_op_B_reg) & (opcode_reg[1] | opcode_reg[2]);
// assign invalid_opcode = opcode_reg[2] & opcode_reg[3]; // Wrong
assign invalid_opcode = opcode_reg[1] & opcode_reg[2]; // Fix
assign invalid = invalid_red_op | invalid_opcode;

always @(posedge clk or posedge rst) begin
  if(rst) begin
    leds <= 0;
    out <= 0;
    cin_reg <= 0;
    red_op_B_reg <= 0;
    red_op_A_reg <= 0;
    bypass_B_reg <= 0;
    bypass_A_reg <= 0;
    direction_reg <= 0;
    serial_in_reg <= 0;
    A_reg <= 0;
    B_reg <= 0;

  end
  else begin
    // if (invalid) // Wrong
    if (invalid && !bypass_A_reg && !bypass_B_reg) // Fix
      leds <= ~leds;
    else
      leds <= 0;

    cin_reg <= cin;
    red_op_B_reg <= red_op_B;
    red_op_A_reg <= red_op_A;
    bypass_B_reg <= bypass_B;
    bypass_A_reg <= bypass_A;
    direction_reg <= direction;
    serial_in_reg <= serial_in;
    opcode_reg <= opcode;
    A_reg <= A;
    B_reg <= B;

    // if (invalid) // Wrong
    if (bypass_A_reg && bypass_B_reg)
      out <= (INPUT_PRIORITY == "A")? A_reg: B_reg;
    else if (bypass_A_reg)
      out <= A_reg;
    else if (bypass_B_reg)
      out <= B_reg;
    else if (invalid)
        out <= 0;
    else begin
        //case (opcode) // wrong
        case (opcode_reg) // FIX
          3'h0: begin // Fix
            if (red_op_A_reg && red_op_B_reg)
              out = (INPUT_PRIORITY == "A")? |A_reg: |B_reg; // FIX
           // out = (INPUT_PRIORITY == "A")? &A_reg: &B_reg; // Wrong
            else if (red_op_A_reg)
              out <= |A_reg; // FIX
           // out <= &A_reg; // Wrong
            else if (red_op_B_reg)
              out <= |B_reg; // FIX
           // out <= &B_reg; // Wrong
            else
              out <= A_reg | B_reg; // FIX
           // out <= A_reg & B_reg; // Wrong
          End
          3'h1: begin // Fix
            if (red_op_A_reg && red_op_B_reg)
              out <= (INPUT_PRIORITY == "A")? ^A_reg: ^B_reg;
           // out <= (INPUT_PRIORITY == "A")? |A_reg: |B_reg;
            else if (red_op_A_reg)
              out <= ^A_reg;
           // out <= |A_reg;
            else if (red_op_B_reg)
              out <= ^B_reg;
           // out <= |B_reg;
            else
              out <= A_reg ^ B_reg;
           // out <= A_reg | B_reg;
          end
          3'h2: begin
            if (FULL_ADDER == "ON")
              out <= A_reg + B_reg + cin_reg;
            else
              out <= A_reg + B_reg;
          end
          3'h3: out <= A_reg * B_reg;
          3'h4: begin
            if (direction_reg)
              out <= {out[4:0], serial_in_reg};
            else
              out <= {serial_in_reg, out[5:1]};
          end
          3'h5: begin
            if (direction_reg)
              out <= {out[4:0], out[5]};
            else
              out <= {out[0], out[5:1]};
          end
        endcase
    end
  end
end

endmodule
```

Assertion:

```
import shared_pkg::*;
`define mk_assert(condition) assert property (@(posedge clk) disable iff(rst) condition)
`define mk_cover(condition) cover property (@(posedge clk) disable iff(rst) condition)
module ALSU_sva(A, B, cin, serial_in, red_op_A, red_op_B, opcode, bypass_A, bypass_B, clk, rst, direction, leds, out);
input bit clk;
input logic rst, cin, red_op_A, red_op_B, bypass_A, bypass_B, direction, serial_in;
input opcode_e opcode;
input logic signed [2:0] A, B;
input logic [15:0] leds;
input logic [5:0] out;

////////////////////////////////////////////////////////
// this variable to make writing assertion more easer //
////////////////////////////////////////////////////////
bit isBypass, InvalidOP, bitwise;
assign isBypass = bypass_A | bypass_B;
assign InvalidOP = opcode==6 | opcode==7;
assign bitwise = opcode==0 | opcode==1;
assign isRed = red_op_B | red_op_A;

///////////////////////////////////
// assertion of bypass operation //
///////////////////////////////////
bypass_A_assert: `mk_assert(bypass_A |-> ##2 out==$past(A,2));
bypass_B_assert: `mk_assert((bypass_B && !bypass_A) |-> ##2 out==$past(B,2));

//////////////////////////////////////
// assertion of reduction operation //
//////////////////////////////////////
sequence redValid(red, op,red2);
    (red && !isBypass && opcode==op && !red2);
endsequence

redA_OR_assert: `mk_assert(redValid(red_op_A,OR,0)        |-> ##2 out==|($past(A,2)));
redB_OR_assert: `mk_assert(redValid(red_op_B,OR,red_op_A) |-> ##2 out==|($past(B,2)));

redA_XR_assert: `mk_assert(redValid(red_op_A,XOR,0)        |-> ##2 out==^($past(A,2)));
redB_XR_assert: `mk_assert(redValid(red_op_B,XOR,red_op_A) |-> ##2 out==^($past(B,2)));

////////////////////////////////
// assertion of invalid cases //
////////////////////////////////
sequence Invalid_seq;
    ((isRed && !isBypass && !bitwise) || (InvalidOP && !isBypass));
endsequence

Invalid_assert: `mk_assert(Invalid_seq |-> ##[0:2](out==0 && leds=='hffff));
```



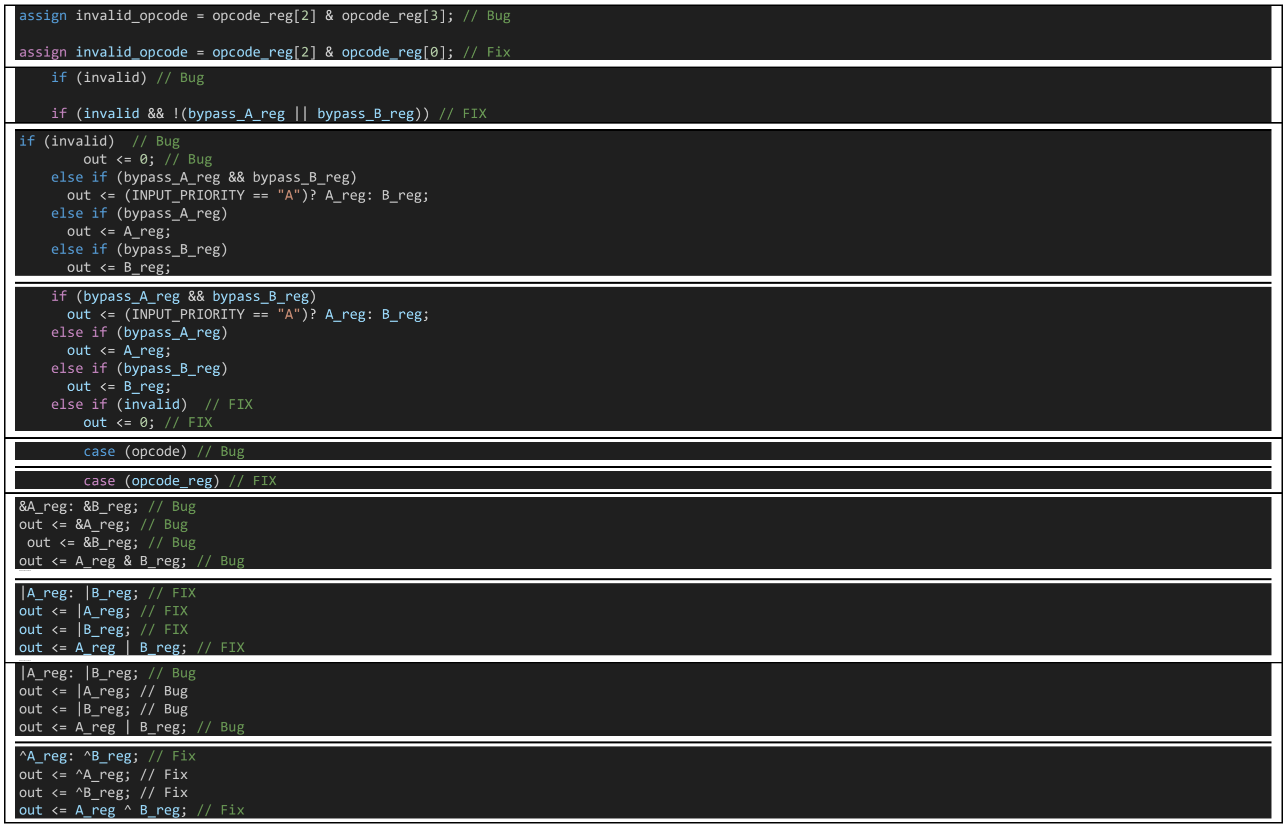
```
////////////////////
// reset assertion //
////////////////////
always_comb begin : reset_assertion
    if (rst) begin
        rst_out_assert: assert final (out==0);
        rst_leds_assert: assert final (leds==0);
    end
end

//////////////////////////////////////////////////////////////////
// assertion for opcode Valid cases without bypass or reduction //
//////////////////////////////////////////////////////////////////
sequence sh_ro(op, dir);
    (opcode==op && dir && !isRed && !isBypass);
endsequence
sequence op_assert(op);
  (!isBypass && !isRed && opcode==op);
endsequence

OR_assert:   `mk_assert(op_assert(0) |-> ##2  out== $past(A,2) | $past(B,2) );
XOR_assert:  `mk_assert(op_assert(1) |-> ##2  out==($past(A,2)^$past(B,2)) );
add_assert:  `mk_assert(op_assert(2) |-> ##2  out==($past(A,2)+$past(B,2)+$past(cin,2)) );
mult_assert: `mk_assert(op_assert(3) |-> ##2  out==($past(A,2)*$past(B,2)) );

shiftL_assert: `mk_assert(sh_ro(4, direction) |-> ##2  out=={$past(out[4:0]), $past(serial_in,2)} );
shiftR_assert: `mk_assert(sh_ro(4, !direction) |-> ##2  out=={$past(serial_in,2), $past(out[5:1])} );

rotateL_assert: `mk_assert(sh_ro(5, direction) |-> ##2  out=={$past(out[4:0]), $past(out[5])} );
rotateR_assert: `mk_assert(sh_ro(5, !direction) |-> ##2  out=={$past(out[0]), $past(out[5:1])} );

/////////////////////////////////////////////////////////////////////////////
//////////////////////////          cover          //////////////////////////
/////////////////////////////////////////////////////////////////////////////

///////////////////////////////
// cover of bypass operation //
///////////////////////////////
bypass_A_cover: `mk_cover( bypass_A |-> ##2 out==$past(A,2));
bypass_B_cover: `mk_cover( (bypass_B && !bypass_A) |-> ##2 out==$past(B,2));

//////////////////////////////////
// cover of reduction operation //
//////////////////////////////////

redA_OR_cover: `mk_cover( redValid(red_op_A,0,0)        |-> ##2 out==|($past(A,2)));
redB_OR_cover: `mk_cover( redValid(red_op_B,0,red_op_A) |-> ##2 out==|($past(B,2)));

redA_XR_cover: `mk_cover( redValid(red_op_A,1,0)        |-> ##2 out==^($past(A,2)));
redB_XR_cover: `mk_cover( redValid(red_op_B,1,red_op_A) |-> ##2 out==^($past(B,2)));

////////////////////////////
// cover of invalid cases //
////////////////////////////
Invalid_cover: `mk_cover( Invalid_seq |-> ##[0:2](out==0 && leds=='hffff));

////////////////
// reset cover //
////////////////
always_comb begin : reset_cover
    if (rst) begin
        rst_out_cover: cover final (out==0);
        rst_leds_cover: cover final (leds==0);
    end
end

//////////////////////////////////////////////////////////////
// cover for opcode Valid cases without bypass or reduction //
//////////////////////////////////////////////////////////////

OR_cover:   `mk_cover( op_assert(0) |-> ##2  out== $past(A,2) | $past(B,2) );
XOR_cover:  `mk_cover( op_assert(1) |-> ##2  out==($past(A,2) ^ $past(B,2)) );
add_cover:  `mk_cover( op_assert(2) |-> ##2  out==($past(A,2) + $past(B,2) + $past(cin,2)) );
mult_cover: `mk_cover( op_assert(3) |-> ##2  out==($past(A,2) * $past(B,2)) );

shiftL_cover: `mk_cover( sh_ro(4, direction) |-> ##2  out=={$past(out[4:0]), $past(serial_in,2)} );
shiftR_cover: `mk_cover( sh_ro(4, !direction) |-> ##2  out=={$past(serial_in,2), $past(out[5:1])} );

rotateL_cover: `mk_cover( sh_ro(5, direction) |-> ##2  out=={$past(out[4:0]), $past(out[5])} );
rotateR_cover: `mk_cover( sh_ro(5, !direction) |-> ##2  out=={$past(out[0]), $past(out[5:1])} );
endmodule
```

Golden model:

```
import shared_pkg::*;

module ALUS_ref (A, B, cin, serial_in, red_op_A, red_op_B, opcode, bypass_A, bypass_B, clk, rst, direction, leds, out);
input clk, rst, cin, red_op_A, red_op_B, bypass_A, bypass_B, direction, serial_in;
input opcode_e opcode;
input signed [2:0] A, B;
output reg [15:0] leds;
output reg [5:0] out;

reg cin_reg, red_op_A_reg, red_op_B_reg, bypass_A_reg, bypass_B_reg, direction_reg, serial_in_reg;
reg [2:0] A_reg, B_reg;
opcode_e opcode_reg;

wire invalid, invalid_opcode, invalid_red;
assign invalid_opcode = (opcode_reg == INVALID_6 || opcode_reg == INVALID_7);
assign invalid_red = (red_op_A_reg || red_op_B_reg) && (opcode_reg != OR && opcode_reg != XOR);
assign invalid = invalid_opcode || invalid_red;

always @(posedge clk or posedge rst) begin
    if (rst) begin
        out <= 0;
        leds <= 0;
        cin_reg <= 0;
        red_op_A_reg <= 0;
        red_op_B_reg <= 0;
        bypass_A_reg <= 0;
        bypass_B_reg <= 0;
        direction_reg <= 0;
        serial_in_reg <= 0;
        A_reg <= 0;
        B_reg <= 0;
    end
    else begin
        if (invalid && !bypass_A_reg && !bypass_B_reg) leds <= ~leds;
        else leds <= 0;

        cin_reg <= cin;
        red_op_A_reg <= red_op_A;
        red_op_B_reg <= red_op_B;
        bypass_A_reg <= bypass_A;
        bypass_B_reg <= bypass_B;
        direction_reg <= direction;
        serial_in_reg <= serial_in;
        opcode_reg <= opcode;
        A_reg <= A;
        B_reg <= B;

        if (bypass_A_reg && bypass_B_reg)
        out <= (INPUT_PRIORITY == "A")? A_reg: B_reg;
        else if (bypass_A_reg)
        out <= A_reg;
        else if (bypass_B_reg)
        out <= B_reg;
        else if (invalid)
            out <= 0;
        else if (opcode_reg == OR) begin
            if (red_op_A_reg && red_op_B_reg)
                out <= (INPUT_PRIORITY=="A")? (|A_reg):(|B_reg);
            else if (red_op_A_reg)
                out <= |A_reg;
            else if (red_op_B_reg)
                out <= |B_reg;
            else
                out <= A_reg|B_reg;
        end
        else if (opcode_reg == XOR) begin
            if (red_op_A_reg && red_op_B_reg)
                out <= (INPUT_PRIORITY=="A")? (^A_reg):(^B_reg);
            else
                out <= (red_op_A_reg)? (^A_reg): (red_op_B_reg)? (^B_reg):(A_reg^B_reg);
        end
        else if (opcode_reg == ADD)
            out <= (FULL_ADDER=="ON")? (A_reg + B_reg + cin_reg):(A_reg + B_reg);
        else if (opcode_reg == MULT)
            out <= A_reg * B_reg;
        else if (opcode_reg == SHIFT) begin
            if (direction_reg)
                out <= {out[4:0], serial_in_reg};
            else
                out <= {serial_in_reg, out[5:1]};
        end
        else if (opcode_reg == ROTATE) begin
            if (direction_reg)
                out <= {out[4:0],out[5]};
            else
                out <= {out[0],out[5:1]};
        end
    end
end
endmodule
```

Bugs Report:

Label	Where in code	Explain	How FIX
invalid_opcode	Line 23	opcode_reg[3] is out of range	Replace opcode_reg[3] With opcode_reg[1]
Blinking Leds	Line42	Bypass is lead signal, If Bypass then we ignore invalid Cases	The condition of (if): Bug: invalid FIX: invalid and not bupass
Checking invaled for out - bypass	Line 56 → 58	Bypass is lead signal, If Bypass then we ignore invalid Cases	Checking for bypass first
Case to check opcode	Line 65	Case should check register opcode not input opcode	case(opcode_reg), instead of: case(opcode)
opcode == 3'h0	Line 68 → 74	When opcode==0, Should OR not AND	Use (|) instead of (&)
opcode == 3'h1	Line 78 → 84	When opcode==1, Should XOR not OR	Use (^) instead of (|)



```
assign invalid_opcode = opcode_reg[2] & opcode_reg[3]; // Bug

assign invalid_opcode = opcode_reg[2] & opcode_reg[0]; // Fix
```

```
    if (invalid) // Bug

    if (invalid && !(bypass_A_reg || bypass_B_reg)) // FIX
```

```
if (invalid)  // Bug
        out <= 0; // Bug
    else if (bypass_A_reg && bypass_B_reg)
      out <= (INPUT_PRIORITY == "A")? A_reg: B_reg;
    else if (bypass_A_reg)
      out <= A_reg;
    else if (bypass_B_reg)
      out <= B_reg;
```

```
    if (bypass_A_reg && bypass_B_reg)
      out <= (INPUT_PRIORITY == "A")? A_reg: B_reg;
    else if (bypass_A_reg)
      out <= A_reg;
    else if (bypass_B_reg)
      out <= B_reg;
    else if (invalid)  // FIX
        out <= 0; // FIX
```

```
        case (opcode) // Bug
```

```
        case (opcode_reg) // FIX
```

```
&A_reg: &B_reg; // Bug
out <= &A_reg; // Bug
 out <= &B_reg; // Bug
out <= A_reg & B_reg; // Bug
```

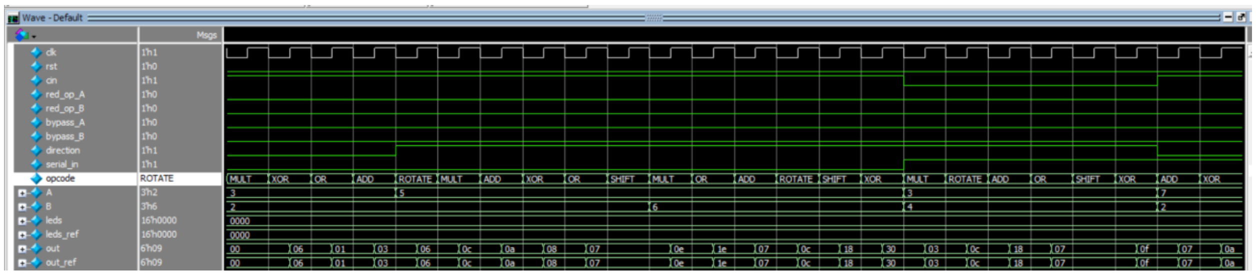
```
|A_reg: |B_reg; // FIX
out <= |A_reg; // FIX
out <= |B_reg; // FIX
out <= A_reg | B_reg; // FIX
```

```
|A_reg: |B_reg; // Bug
out <= |A_reg; // Bug
out <= |B_reg; // Bug
out <= A_reg | B_reg; // Bug
```

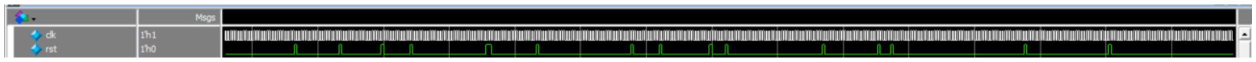
```
^A_reg: ^B_reg; // Fix
out <= ^A_reg; // Fix
out <= ^B_reg; // Fix
out <= A_reg ^ B_reg; // Fix
```

QuestaSim Snippet:

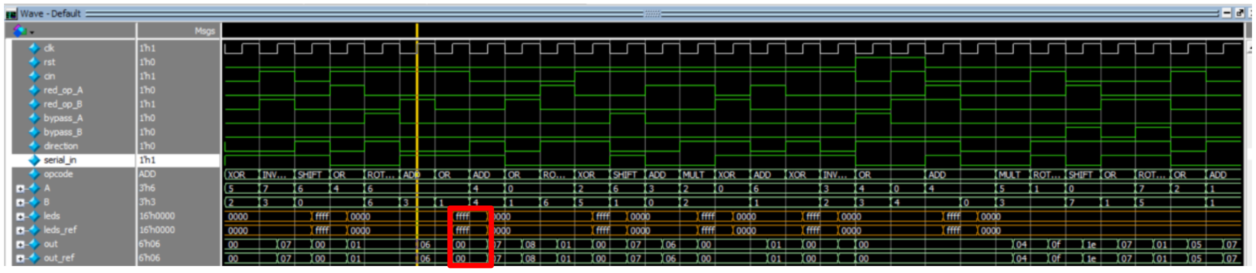
Constraint rule_8 No invalid cases



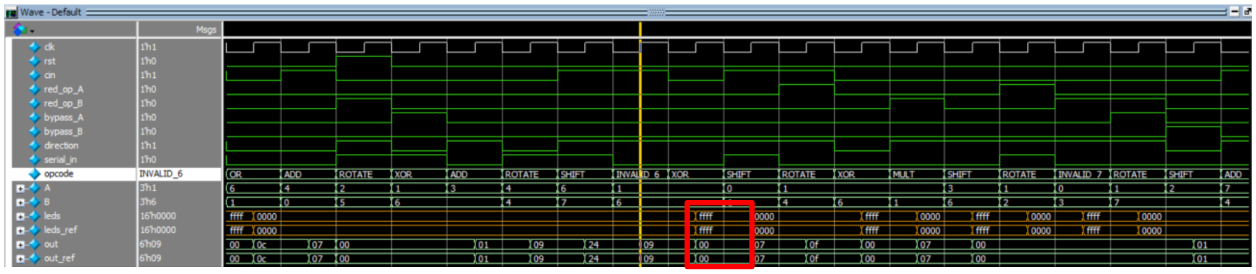
rst constraint most of the time Inactive



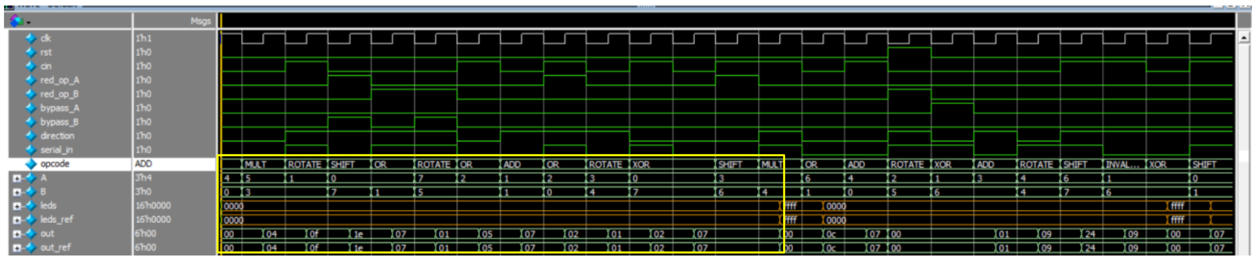
red_op_A is active while opcode is ADD ➔ Invalid case ➔ leds blink and out = zero



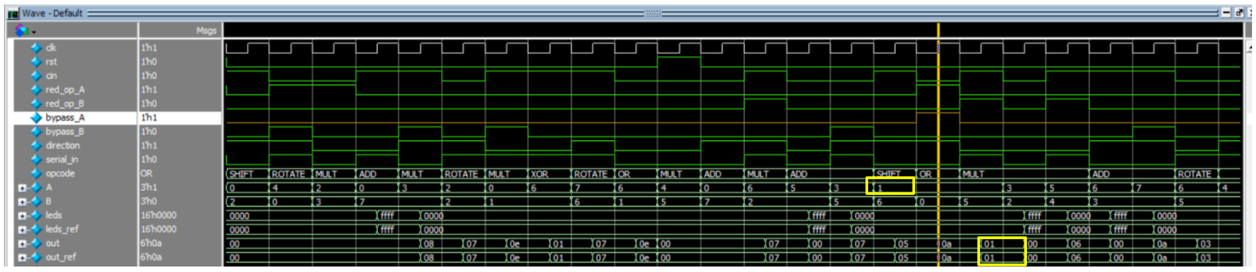
Opcode = Invalid case



Valid Opcode Cases

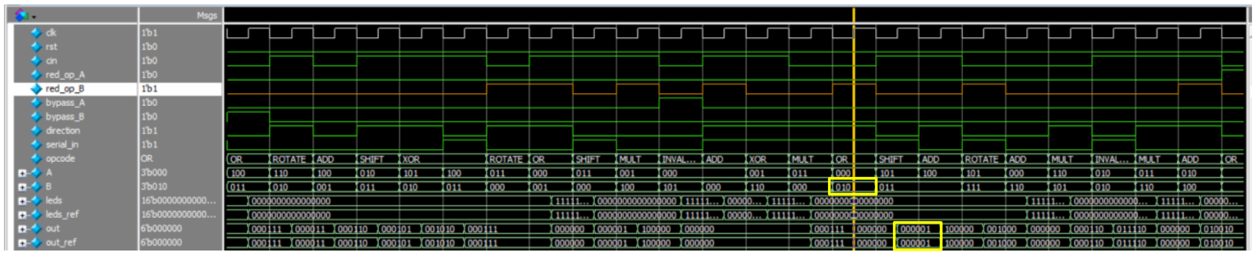


Bypass_A



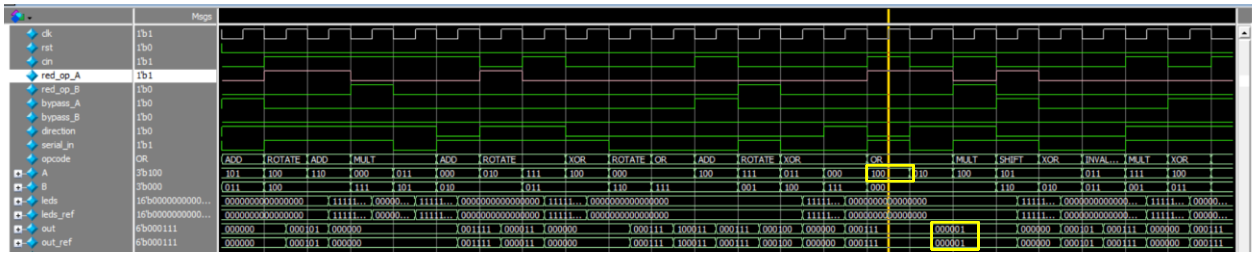
```
clk        1'h1
rst        1'h0
cin        1'h0
red_op_A   1'h1
red_op_B   1'h0
bypass_A   1'h1
bypass_B   1'h0
direction  1'h1
serial_in  1'h0
opcode     OR
A          3'h1
B          3'h0
leds       16'h0000
leds_ref   16'h0000
out        6'h0a
out_ref    6'h0a
```

Opcode = OR, red_op_B active



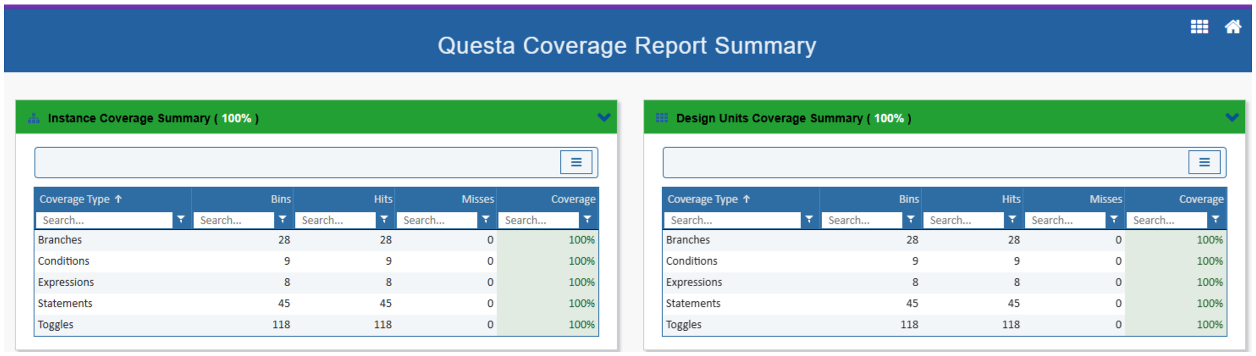
```
clk        1'b1
rst        1'b0
cin        1'b0
red_op_A   1'b0
red_op_B   1'b1
bypass_A   1'b0
bypass_B   1'b0
direction  1'b1
serial_in  1'b1
opcode     OR
A          3'b000
B          3'b010
leds       16'b0000000000...
leds_ref   16'b0000000000...
out        6'b000000
out_ref    6'b000000
```

Opcode = OR, red_op_A active



```
clk        1'b1
rst        1'b0
cin        1'b1
red_op_A   1'b1
red_op_B   1'b0
bypass_A   1'b0
bypass_B   1'b0
direction  1'b0
serial_in  1'b1
opcode     OR
A          3'b100
B          3'b000
leds       16'b0000000000...
leds_ref   16'b0000000000...
out        6'b000111
out_ref    6'b000111
```

Code Coverage HTML report



Questa Coverage Report Summary

Instance Coverage Summary (100%)

Coverage Type	Bins	Hits	Misses	Coverage
Branches	28	28	0	100%
Conditions	9	9	0	100%
Expressions	8	8	0	100%
Statements	45	45	0	100%
Toggles	118	118	0	100%

Design Units Coverage Summary (100%)

Coverage Type	Bins	Hits	Misses	Coverage
Branches	28	28	0	100%
Conditions	9	9	0	100%
Expressions	8	8	0	100%
Statements	45	45	0	100%
Toggles	118	118	0	100%

Code Coverage txt report

- Branches

```
Branch Coverage:
    Enabled Coverage                Bins      Hits    Misses  Coverage
    ----------------                ----      ----    ------  --------
    Branches                          28        28         0   100.00%
```

- Conditions

```
Condition Coverage:
    Enabled Coverage              Bins   Covered    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Conditions                       9         9         0   100.00%
```

- Expression

```
Expression Coverage:
    Enabled Coverage              Bins   Covered    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Expressions                      8         8         0   100.00%
```

- Statement

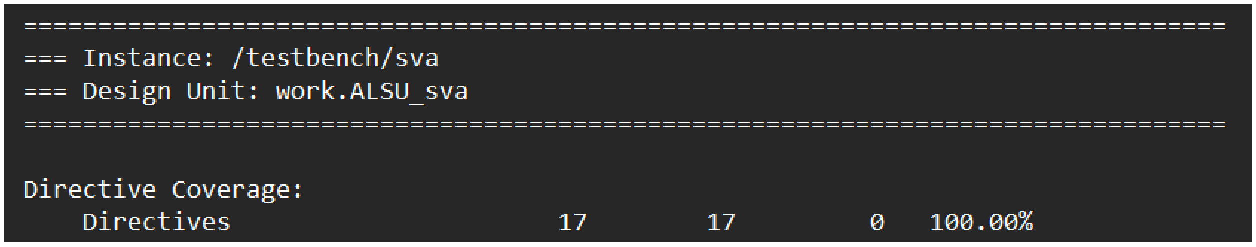
```
Statement Coverage:
    Enabled Coverage                Bins      Hits    Misses  Coverage
    ----------------                ----      ----    ------  --------
    Statements                        45        45         0   100.00%
```

- Toggles

```
Toggle Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Toggles                        118       118         0   100.00%
```

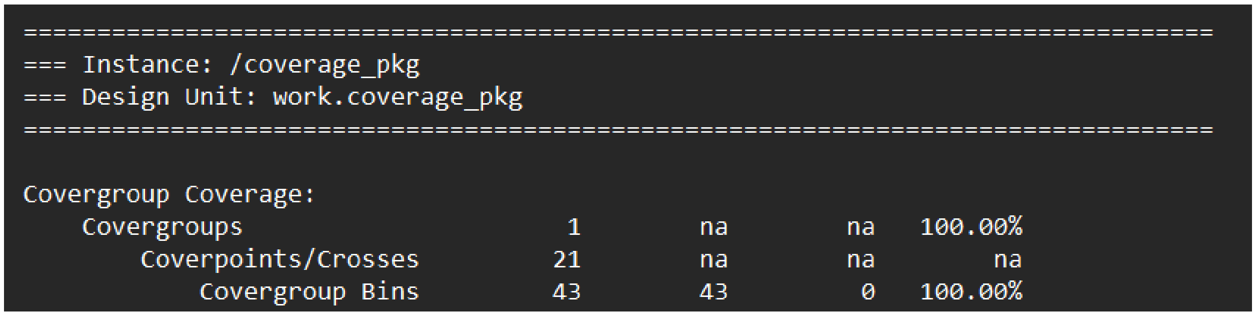
FUNCTION_COVER report

- Directive Coverage



```
=================================================================================
=== Instance: /testbench/sva
=== Design Unit: work.ALSU_sva
=================================================================================

Directive Coverage:
    Directives                      17        17         0   100.00%
```

- Covergroup Coverage



```
=================================================================================
=== Instance: /coverage_pkg
=== Design Unit: work.coverage_pkg
=================================================================================

Covergroup Coverage:
    Covergroups                      1        na        na   100.00%
        Coverpoints/Crosses         21        na        na        na
            Covergroup Bins         43        43         0   100.00%
```